Facilities for testing control software

Pieter J. Schoenmakers
<tiggr@ics.ele.tue.nl>

Jochen A.G. Jess
<jess@ics.ele.tue.nl>

Department of Electrical Engineering
Eindhoven University of Technology
P.O.Box 513, 5600 MB Eindhoven, the Netherlands

Abstract

In this paper we outline an approach to the development and testing of control software for embedded systems. This approach enables system level testing of the software without the hardware with which it normally interacts. This independence implies that testing can commence prior to hardware availability and it enables unattended, frequent test runs.

The proposed method is a generalization of existing ad hoc solutions to similar testing problems. We observe the requirements underlying this generalization and present an object oriented programming language that meets these requirements.

1 Introduction

In large co-designed embedded systems two kinds of software can be discerned. One kind is largely concerned with algorithmic data processing; the other kind is the software that handles the control of the whole system. This control software can be very complex; it is the result of a design effort by a possibly large team. In a concurrent design process, this team faces the problem of testing the control software while the hardware system it will run on is not yet available. In the project's maintenance phase, hardware will be available but testing every corner of the software while depending on the actual hardware might not be a feasible option. All in all, it is desirable, if not mandatory, to be able to test the control software independent of the hardware system.

In this paper we propose a general method for testing control software independent of the hardware, which

- supports real time load and stress tests,
- enables testing at higher levels of abstraction than plain hardware interfacing, and
- can provide hardware designs with realistic test benches.

More specifically, we describe the general facilities that must be offered by the test environment and the system level specification language to support this method. A major goal of our work is to enable unattended test runs of load and stress tests at the system and subsystem level. Unattended runs enable (frequent) regression testing, which increases the confidence in the system's correct operation.

In section 2 we compare our testing method with existing approaches; section 3 outlines the method. Section 4 defines the requirements of the development environment in support of the method; section 5 introduces the environment's programming language. Before concluding, we present the results of applying the method to the example of an elevator control.

2 Testing

Testing a piece of software consists of several steps: the part of the system to be tested must be identified, test cases must be designed, served as input to the system under test (SUT), and their outcome must be verified against the expected results. Numerous methods addressing these testing issues already exist [1] and the testing method presented in this paper will not increase their number. Instead, we develop a method that enables the application of these software testing methods to situations where the software cannot meaningfully operate without the hardware with which it interacts.

In effect, the testing method proposed in this paper is a generalization of ad hoc solutions to testing software independent of the hardware. An example of such a solution is testing a program on/through its graphical user interface (GUI). In this case, the hardware to be eliminated consists of the mouse, keyboard and screen. The various ways of GUI testing all have in common that, at some level of abstraction, user control of the mouse and keyboard is run by a test driver, and, also at varying levels of abstraction, the program output or state is checked against the expected results as defined for the test. Generalizing these mechanisms is a major goal of our work.

3 The method

Testing the software independent of the hardware requires that during testing the hardware is replaced by a model of the hardware. This model can e.g. be provided by a hardware emulator, a simulator running a description of the hardware, but also purely by software written for this purpose.

In general, during normal operation, the interfaces between hardware and software consist of pieces of software which provide the control software with an abstraction of the hardware. We use the term *hardware abstractions*[1] to denote these interfaces (figure 1a). During test runs, each abstraction of hardware must instead provide an abstraction of a *model of the hardware.*

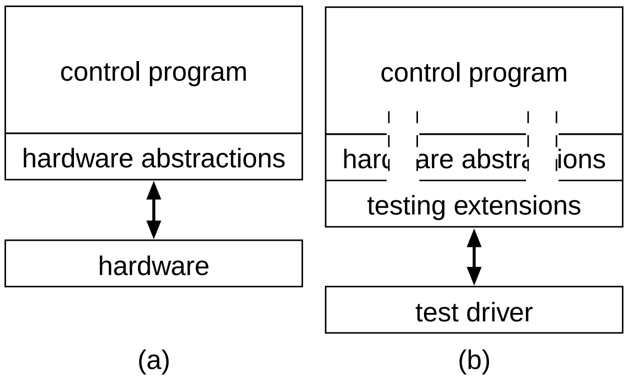

Figure 1: *normal operation (a), and operation during testing (b).*

While testing, the software to be tested (the *system under test* or SUT) is controlled by a test driver. The test driver is responsible for executing the tests and verifying their outcome. To make the SUT ready for execution, it is extended with *testing extensions* which change the SUT's hardware abstractions into abstractions of the hardware models, making the SUT independent of the hardware on which it normally depends (figure 1b). The hardware models then provide the SUT with its inputs. These models in turn receive directions from the *simulated world* in which they operate. The simulated world is maintained by the test driver, which can be customized to suit the SUT.

[1]We use 'hardware abstraction' instead of the widely used 'driver' to avoid confusion with 'test driver'.

The test driver program provides a framework in which the simulated world can be created and scenarios can be generated and, if applicable, recorded and played back. Tests can be verified by comparing the outcome against the outcome dictated by the scenario or by checking the system's behavior against sets of invariants and postconditions. These conditions can apply to the SUT, parts thereof, or parts of the simulated world.

An important advantage of modeling the hardware purely by software is that this enables modeling at higher levels of abstraction than plain interfaces. For example, checking the contents of a text widget is much easier than checking its appearance on the screen, and interacting with a program by issuing events is much easier than simulating the mouse and keyboard. Another advantage is that the hardware models only need to be as good as necessary to achieve the testing goal. For example, when doing basic transaction testing on an elevator control, only the scheduling subsystem is being tested and timing information can be safely omitted from the models. Only when the tests are extended to include information on time to service, the models must take into account the time it takes for the cage to move and for passengers to enter and exit. In effect this means that the testing models can be developed in parallel with the system: new models can be included as each subsystem is implemented and existing models can be refined when the needs of testing dictate so.

It is possible that the outcome of the tests cannot be verified by observing the behavior of the hardware models. In such cases the testing extensions must include functionality to observe the SUT's internal state. The testing extensions must also enable the test driver to set the SUT's state at the start of a test. The extensions should of course not alter the system's behavior as that would invalidate the test results or, even worse, violate the truthfulness of the tests.

It is important to note that the testing extensions are not an inherent part of the system's functionality. These extensions are absent during normal operation; they are not part of the system's design. If they were, they would unnecessarily increase the complexity of the system's specification. This observation signifies the difference between design for test (DfT) for hardware and for software: with hardware DfT, specific facilities are incorporated into the design to gain access to the hardware's internal state, whereas with software, access is essentially unrestricted, were it not for the specification language or paradigm used.

The target machine on which the software is to run might not be capable of supporting the software plus testing extensions, for instance because it is too small to include a test driver and it does not support network communication which would enable running the test driver on another machine. The testing process must therefore be independent of the machine on which testing is performed. To support real time testing, the test driver must support distinctive notions of elapsed simulated time, CPU time, and wall time. For low precision timing, simulated time can be bound to CPU time or even wall time. A higher accuracy of simulated time is obtained by decoupling it from wall time. When this does not suffice, simulated time must be made independent of CPU time, e.g. by accurate code instrumentation.

With simulated time being unrelated to wall time, simulated time can be sped up to decrease the time needed to run the tests, and it can be slowed down if time is needed to compute a hardware model. Since the testing process is independent of the machine on which testing takes place, a suitable machine can be chosen on which, for instance, coverage and profiling tools are available.

4 Facilities

To support the testing method outlined above, the following facilities are required:

- the possibility to extend the control software so as to make it suitable for execution under the test driver, i.e. to change the hardware abstractions into abstractions of models of the hardware, and

- the possibility to extend the software in order to make its internal state available to the test driver, and to have the driver set the system's state.

These requirements essentially state that it must be possible to extend an existing program without those extensions being part of the program's design. This extensibility has the advantage that the program and its testing extensions can be developed independently. In addition, some interesting advantages are offered if extensibility does not require recompilation:

- older versions of the program can be tested against the latest test extensions, without requiring retrieval and recompilation of the old version of the sources, and
- testing extensions can always be developed and applied, even in case (part of the) source code is unavailable.

5 Implementation

To evaluate the proposed testing method, we have developed an object oriented (OO) programming language to be used as the language in which to develop the control software. The required testing facilities imply that the language must enable (1) modification of the behavior of the objects implementing the hardware abstractions, and (2) addition of behavior to any object, to at least retrieve or set its internal state. In a single sentence: the behavior of objects must be amendable; this means that the way in which they respond to method invocations can be changed.

We have designed and implemented TOM: an object oriented programming language which supports the testing paradigm presented in this paper. To this end, TOM provides the following facilities:

- classes are extensible entities: a class is fully defined by its main definition and any extensions defined for it,
- an extension can add and replace methods. In support of complex added behavior, an extension can add instance variables. To promote reuse, an extension can introduce additional superclasses, and
- extensions can be added to a program at compile, link, or run time.

TOM is an object oriented programming language in the spirit of Objective-C [2] and similar to Java [3] in its deviation from the original language. In the remainder of this section, we address some of its properties that make it suitable for the development of (control) software.

All objects reside on the heap that is managed by an incremental tracing garbage collector. The garbage collector can be invoked with a limit on the elapsed time during its run. This limit can be lowered to decrease the latency in the program's response to events. Automatic invocation of the garbage collector can be inhibited during time critical sections.

Every method invocation is dynamically bound. This is required to be able to replace methods, since the method invoked by a statically bound or inlined method invocation cannot be overridden.

Direct access to instance variables is only possible for the current object, `self`. Access to the variables of another object must employ a method invocation. This introduces some overhead, but ensures that to adjust the meaning or usage of a certain variable, a testing extension needs to replace only a single method.

TOM discerns primitive types from objects. These primitive types include integer and floating point types. Values of these types are always passed by value, as opposed to by reference, and operations on them are statically bound. In short, such primitive values are handled fast by a processor. This is important for keeping a program's execution time at acceptable levels.

The overhead of dynamic method binding accounts for approximately 5 to 10% of the CPU time, depending on the program. Garbage collection consumes up to 10%, depending on the rate at which the program allocates objects and disposes of them, and the frequency of garbage collector runs (more frequent runs in general reduce a program's memory footprint at the expense of CPU time). These times do not pose a problem for the kind of applications for which TOM was designed.

It is important to mention that the development of programs in Objective-C is reported to take considerably less time than the development of similar programs in a language like C++, though no references to any precise measurements are available. The Objective-C advantage is generally attributed to its extensive employment of dynamic binding and typing. TOM outperforms Objective-C in this respect and is expected to also exhibit this advantage. This means that program development will take less time and for control software of an embedded system, a decrease in development time is generally more important than a slight increase in a program's execution time.

The interested reader is referred to [4] for more information on TOM.

6 Application

We have applied the proposed testing method to an elevator control program. In this section, we sketch the implementation and results. For a more elaborate discussion, see [5].

The elevator control program controls a system consisting of a number of elevators which must service some number of floors. All elevators are equal, and a request at a floor will be serviced by one of them.

The program employs several classes of objects that provide an abstraction of physical entities such as the engines, displays, and switches. In normal usage, the example program employs a bidirectional stream of bytes to interact with the hardware. Input from that stream is handled by a parser, which decodes the input events and invokes the proper method of the proper object abstracting the hardware from which the event originated. In the output direction, a command issued over the stream is decoded and acted upon by the piece of hardware that is identified by the command.

Commands are issued from the program by specific methods of the hardware abstracting objects. Engine commands, for example, are only issued from the 'instructEngine' method of the Elevator class; moreover, this method performs no other action. Such localization of interface to the hardware abstraction is the only concession to testability that is present in the design of the control program. It is not strictly necessary, as the testing extension could duplicate any extra actions of the overridden method. Localization is, however, convenient.

6.1 Test extension

In the example testing situation we are interested in proper operation of the elevator system and scheduler, in its response to its inputs. This interface consists of the switches in each elevator, on each floor, and the guidance switches in each elevator shaft. We check for proper operation by observing the time spent by users waiting for an elevator to arrive and traveling, by elevator, to their destination.

In the proposed setup, several subsystems are not tested, including door management and information display. The test could be extended to include these as well; for our example testing goals they are not needed.

In the simulated world, an elevator is modeled by a cage that travels at constant speed, with infinite acceleration and deceleration. The cage travels within a shaft with all switches and floors at the appropriate places. The simulated world is completed by virtual persons which exhibit some semi-random behavior and interact with the system by pushing switches.

The test extension applies the following modifications to the classes of the elevator control program to make it suitable for running tests:

- Every command issuing method is replaced by a method that triggers similar functionality in the simulated world.
- The Elevator and Floor classes are modified so that each instance maintains a set of persons that observe the instance. The behavior that is common is added by inheritance of an extra superclass.
- Several methods are replaced by empty methods to ensure that the hardware interfaces which are unused during testing, actually do nothing.
- Several methods are added to various classes to provide direct (modifying) access to the objects' internal state.
- A Person class is introduced to model the persons inhabiting the simulated world and provide the system with its inputs.

In the testing situation, with this very simple test driver, simulated time runs synchronous to wall time while executing code. Running the tests on a UNIX machine with other processes occasionally needing the processor's attention, a precise (deep sub-second) notion of simulated time cannot be maintained. On one hand, this has the advantage of free non-determinism. On the other hand, checking the test results cannot be as simple as the comparison of an event trace with known expected output. Correct operation of the system is therefore checked by code; if none of these checks fail, the system passes the test.

6.2 Time

Most of the time the elevator system is idle, waiting for some switch to be hit. In simulated time, waiting is necessary of course. In wall time however, waiting is wasting time. This problem is solved by changing the program's notion of time whenever it is about to start waiting: when it needs to wait for x seconds, time is simply advanced by x. The first scheduled event will then immediately trigger.

A similar problem with time is caused by lengthy operations such as printing debug or log information. In this case, the problem is solved by restoring time afterwards to what it was before printing started.

Another difference between normal operation and the testing situation is that during normal operation method pre- and postcondition checking is usually disabled, whereas they are usually enabled during test runs. The side-effect of the time consumed by these checks cannot be undone. Luckily however, their influence is no more significant than that of the different machine on which the tests are run compared to the normal situation.

With time manipulation performed as outlined above, we can test the system in only a fraction of the simulated time. For example, on a 180MHz PA-8000 in a HP9000/879, simulation of a system with 4 elevators servicing 20 floors and 20 persons for 24 hours completes in less than 150 seconds.

Execution time will obviously increase with the quality of the test bench and the amount of administration needed, but even a tenfold increase is insignificant while maintaining the advantage of hardware independence.

7 Conclusions and Future Work

A major contribution of the method presented in this paper is how existing software testing techniques can be applied to embedded control software

- while not influencing the design of the system to be tested (apart from prerequisites such as the programming paradigm or language used),
- at the same abstraction level as the one at which the program was developed, and
- without needing to revert to using a debugger or other dirty testing methods.

The ideas presented in this paper are just that—ideas. They have been tested on the relatively simple example of a single-threaded event-driven soft real-time program. For use in real applications, a test driver and supportive framework must be developed. Before this can be accomplished, other programming models should be explored and the viability of the testing approach must be assessed for harder real-time systems.

References

[1] Boris Beizer, *Software testing techniques*, Van Nostrand Reinhold, 1990.

[2] NeXT Software, *Object oriented programming and the Objective-C language*, Addison-Wesley, 1993.

[3] James Gosling, Bill Joy, Guy Steele, *The Java language specification*, Addison-Wesley, 1996.

[4] Pieter J. Schoenmakers, *The* TOM *programming language*, http://tom.ics.ele.tue.nl:8080/.

[5] Pieter J. Schoenmakers, "Testing software suffering from hardware", *proceedings of the ProRISC workshop on circuits, systems and signal processing*, Vol. 8, to appear November 1997 .